游易-程序主题分享合辑

知识管理部 等 2022.06.07 15:37 6220 321 184个资源

汇总从101至今的游易征稿文章合辑。

卡通渲染中的描边技术及其原理

郑杰 2021.03.11 11:37 737 14 0 查看原文

> 对于非真实感渲染（Non-photorealistic rendering，简称NPR），现在非真实感渲染领域应用最广的渲染技术是像日本动画的卡通渲染风格，目前一般称之为Cel Shading or Toon Shading，对于这一类的渲染在日本那边很早就在主机游戏上使用，过了很多尝试和变迁，最终在《GUILTY GEAR Xrd》系列游戏达到了非常不错的水准，市面上几乎所有的卡通渲染向制作以GGX shader作为标杆和借鉴。对于卡通渲染来说，对于最终呈现效果最大的两点工作就在于描边处理和艺术化着色

1 序言

对于非真实感渲染（Non-photorealistic rendering，简称NPR），现在非真实感渲染领域应用最广的渲染技术是像日本动画那样的卡通风格，目前一般称之为Cel Shading or Toon Shading，对于这一类的渲染在日本那边很早就在主机游戏上使用，经过了很多尝试和变迁，《GUILTY GEAR Xrd》系列游戏达到了非常不错的水准，市面上几乎所有的卡通渲染向制作都是以GGX shader作为标杆和借鉴。对于卡通渲染来说，对于最终呈现效果最大的两点工作就在于**描边处理**和**艺术化着色**,本文就着重于描边处理方面展开分析

2 描边的技术分类

在Real-Time Rendering一书中将描边技术分为了以下五大类：

1. 基于法线和视角的描边(Shading Normal Contour Edges)
2. 过程式的几何描边(Procedural Geometry Silhouetting)
3. 基于图片处理的描边(Edge Detection by Image Processing)
4. 基于轮廓线检测的描边(Geometric Contour Edge Detection)
5. 混合以上几种描边方法(Hybrid Silhouetting)

基于法线和视角的描边

这种描边算法主要是使用视角方向和法线方向的点积结果float vdotn = dot(viewDir, normal来获取轮廓线的信息。通过观察于边缘的像素点与视角的夹角接近90°，而越远离边缘的像素点与视线的夹角越小。该描边方法既可以通过一个阈值参数_O度

```
half edge = saturate(dot (o.Normal, normalize(IN.viewDir)));
edge = edge < _Outline ? edge/4 : 1;
o.Albedo = (floor(c.rgb * _Tooniness)/_Tooniness) * edge;
```

这种方法最大的优点就是实现简单，但有一个明显的弊端就是，对于平坦、棱角分明的物体，使用上述描边方法会产生突变或非预期的
现线条粗细不一的情况

过程式的几何描边

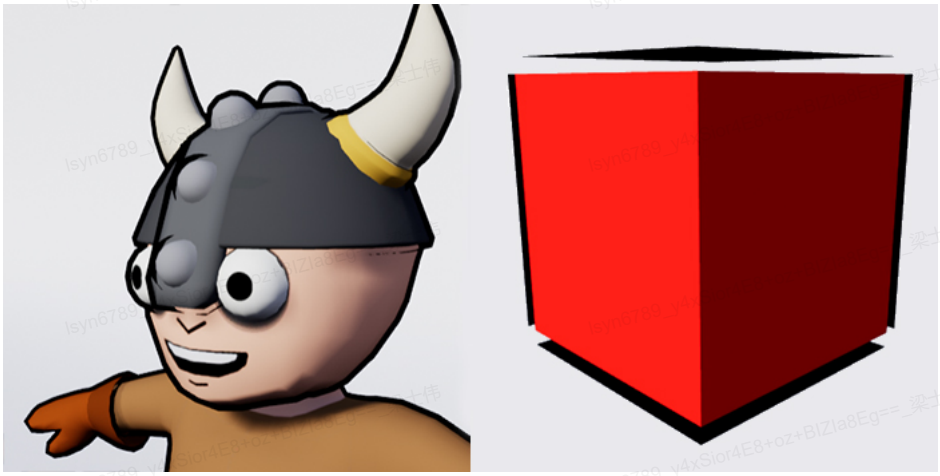
过程式几何描边需要使用两个Pass来进行描边处理。在第一个Pass中只渲染背面的面片，用来实现描边的效果；然后第二个Pass中对模
常的渲染。
通过2次绘制来绘制描边的方法。在《GUILTY GEAR Xrd》中称其为Back Facing法。
第一次绘制角色，第二次绘制描边。绘制描边的时候，在顶点着色器将顶点沿着法线方向位移一段距离，使得模型轮廓放大，渲染作为
时描边绘制时使用cull front。这样描边和角色重叠的部分会因为不能通过深度检测而cull掉，保证描边不会遮挡角色。

```
// 顶点膨胀
v2f vert (a2v v)
{
    v2f o;
    UNITY_INITIALIZE_OUTPUT(v2f, o);
    o.pos = UnityObjectToClipPos(float4(v.vertex.xyz + v.normal * _OutlineWidth * 0.1 ,1));//顶点沿着法线方向外扩
    return o;
}
```

此时有一个问题就是随着摄像机拉进拉远，描边的粗细也会改变，要解决这个问题还需将法线外扩的大小改为使用NDC空间距离外扩

```
v2f vert (a2v v)
{
    v2f o;
    UNITY_INITIALIZE_OUTPUT(v2f, o);
    float4 pos = UnityObjectToClipPos(v.vertex);
    float3 viewNormal = mul((float3x3)UNITY_MATRIX_IT_MV, v.normal.xyz);
    float3 ndcNormal = normalize(TransformViewToProjection(viewNormal.xyz)) * pos.w;//将法线变换到NDC空间
    pos.xy += 0.01 * _OutlineWidth * ndcNormal.xy;
    o.pos = pos;
    return o;
}
```

这种描边的好处是你能得到一个均匀的描边效果并对大部分模型有效，同时也可以通过移动顶点简单的修改调整描边的粗细，缺点就是
描绘面片内部细节，由于描边由多边形组成，它们容易被裁剪，并且要通过两个pass渲染对性能不友好。
最严重的问题就是由于每个面的顶点的法线都垂直于这个平面。所以描边的外扩也是垂直于平面，当模型有转角的情况下，描边就会裂
只比较好适用于表面光滑的模型或凸面体。不过对于这一点现在也有了一个较好的解决方法



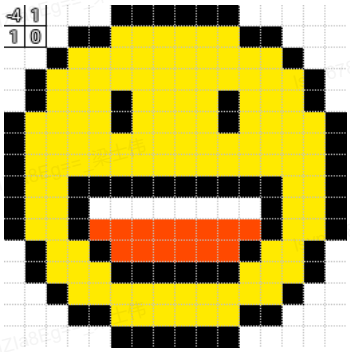
要解决这个问题就是要将模型外扩使用的法线数据进行修改，要将相同位置顶点的法线数据进行平均计算，并将新算出来的法线数据写
的切线数据中，并用这个切线数据进行法线外扩。为什么是写入切线数据是因为只有法线和切线会随骨骼动画改变。

```
private static void WirteAverageNormalToTangent(Mesh mesh)
{
    var averageNormalHash = new Dictionary<Vector3, Vector3>();
    for (var j = 0; j < mesh.vertexCount; j++)
    {
        if (!averageNormalHash.ContainsKey(mesh.vertices[j]))
        {
            averageNormalHash.Add(mesh.vertices[j], mesh.normals[j]);
        }
        else
        {
            averageNormalHash[mesh.vertices[j]] =
                (averageNormalHash[mesh.vertices[j]] + mesh.normals[j]).normalized;
        }
    }

    var averageNormals = new Vector3[mesh.vertexCount];
    for (var j = 0; j < mesh.vertexCount; j++)
    {
        averageNormals[j] = averageNormalHash[mesh.vertices[j]];
    }

    var tangents = new Vector4[mesh.vertexCount];
    for (var j = 0; j < mesh.vertexCount; j++)
    {
        tangents[j] = new Vector4(averageNormals[j].x, averageNormals[j].y, averageNormals[j].z, 0);
    }
    mesh.tangents = tangents;
}
```

不过这种方法是在运行时计算与入，正确的处理方法应该是将数据写入到模型数据中，比如用FBX的SDK工具将计算出的切线数据写入到
为了美术能够更好的订制模型的描边效果，而在《GUILTY GEAR Xrd》中还提出使用模型顶点颜色的四个通道，对模型描边的粗细、显
距离缩放等进行了精细的控制

基于图像处理的方法

这种描边方法就是在图像后处理(Post Process)的时候使用边缘检测算法进行描边绘制，一般用于屏幕空间的检测。我们通常使用深度、
度和颜色等属性作为边缘判断的依据，然后利用边缘检测算子进行边缘的判断。常见的边缘检测算子有：Sobel算子、Canny算子、Lap
子、Robert算子和Prewitt算子。
使用图像检测方法描边的好处就是可以轻松的绘制一整个场景的描边，对性能友好，描边的宽度均匀并且不会被裁切
不好的地方就是需要使用多种边缘检测方法去获取到边界信息，边缘可能会出现变化较大的区域
对于边缘检测最常用的办法就是对每个像素进行卷积操作
对于图像处理来说，卷积操作就是用两组不同的数值计算出一个一个数值出来。所谓的算子就是一组网格数值，通过移动算子去计算出
心像素的卷积值得到我们的输出



3种常见的边缘检测算子如图所示，它们都包含了两个方向的卷积核，分别用于检测水平方向和竖直方向上的边缘信息。

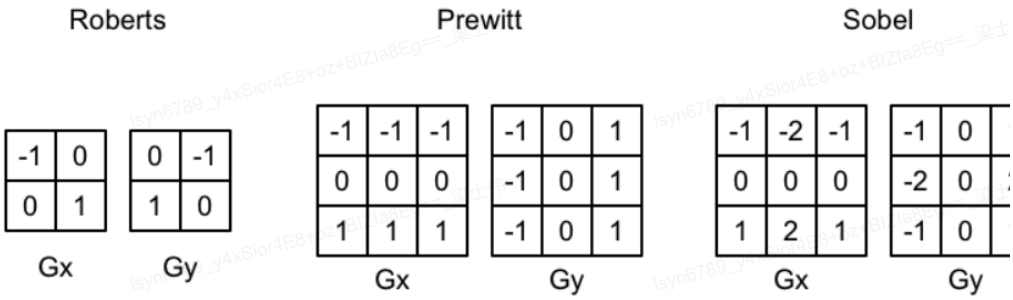


图12.5 三种常见的边缘检测算子

在进行边缘检测时，我们需要对每个像素分别进行一次卷积计算，得到两个方向上的梯度值Gx和Gy，而整体的梯度可按下面的公式计算

$$G = \sqrt{G_x^2 + G_y^2}$$

由于上述计算包含了开根号操作，出于性能的考虑，我们有时会使用绝对值操作来代替开根号操作：

$$G = |G_x| + |G_y|$$

使用图像后处理方法实现描边效果要较为复杂一点，这里提供一个思路就是让所有需要描边的物体在渲染的时候，将 Stencil 参考值写入
Buffer 中。全部写入完成之后，我们就把 Stencil Buffer 提取出来转换成图像，并使得图像上只有 Stencil 值的地方有颜色。然后把这张
屏幕后处理Shader中，根据 Sobel 边缘检测算法对其边缘检测，检测出边缘后与原屏幕图像进行叠加就完成了。

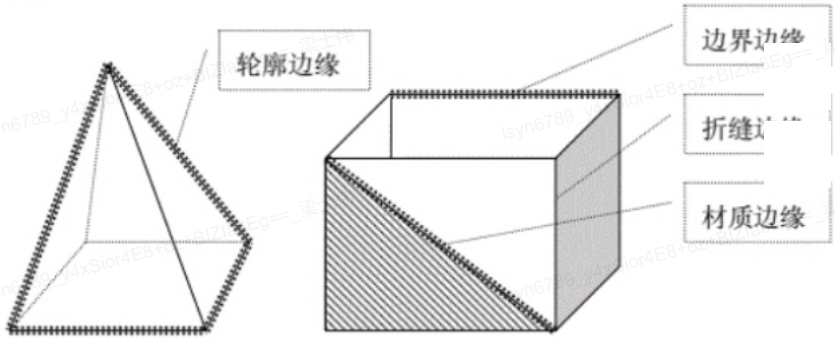
```
struct v2f
{
    float2 uv[9] : TEXCOORD0;   // 算子是3X3的网格
    float4 pos : SV_POSITION;
};
v2f vert (appdata_img v)
{
    v2f o;
    o.pos = UnityObjectToClipPos(v.vertex);
    half2 uv = v.texcoord;

    // 取像素周围一圈的值
```

```
    o.uv[6] = uv + _StencilBufferToColor_TexelSize.xy * half2(-1, 1);
    o.uv[7] = uv + _StencilBufferToColor_TexelSize.xy * half2(0, 1);
    o.uv[8] = uv + _StencilBufferToColor_TexelSize.xy * half2(1, 1);

    return o;
}

float SobelEdge(v2f i)
{

    const half Gx[9] = {-1,  0,  1,
                        -2,  0,  2,
                        -1,  0,  1};
    const half Gy[9] = {-1, -2, -1,
                         0,  0,  0,
                         1,  2,  1};

    float edge = 0;
    float edgeY = 0;
    float edgeX = 0;
    float luminance =0;
    for(int it=0; it<9; it++){
        luminance = tex2D(_StencilBufferToColor,i.uv[it]).a;
        edgeX += luminance*Gx[it];
        edgeY += luminance*Gy[it];
    }

    edge =  1 - abs(edgeX) - abs(edgeY);
    return edge;
}

fixed4 frag (v2f i) : SV_Target
{
    fixed4 sourceColor = tex2D(_MainTex, i.uv[4]);
    float edge = SobelEdge(i);
    return lerp(_EdgeColor,sourceColor,edge);
}
```

3 更高级的描边

从学术方面来考虑，基于空间的3D模型的边缘分为四种：轮廓边、边界边、折缝边、材质边，其中前三种是卡通渲染需要的

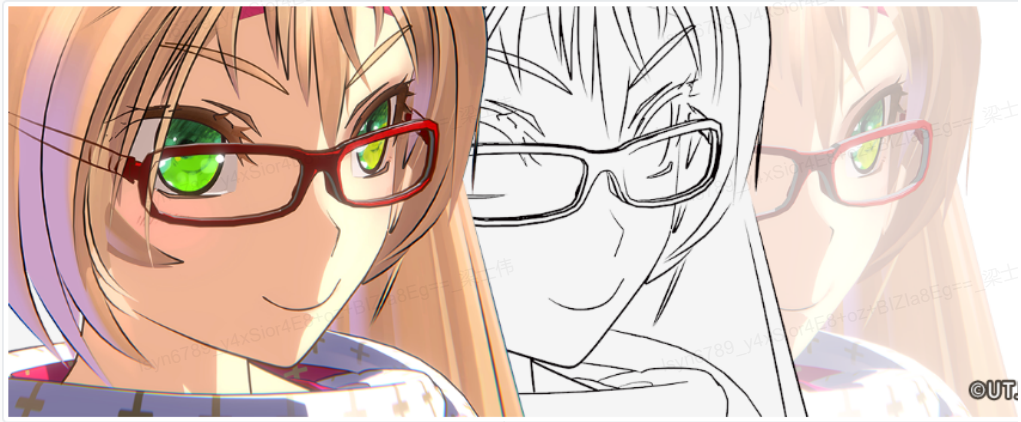
- 轮廓边缘（silhouette edge） 即相邻两个三角形具有不同朝向的边缘。
- 边界边缘（border edge） 即不为两个三角形共享的边缘。比如一张纸的边缘。
- 折缝边缘（crease edge） 由两个三角形共享，并且这两个三角形法线之间的夹角小于某个大小，如 60°。
- 材质边缘（material edge） 即相邻的两个三角形具有不同的材质或者颜色等属性时行成的边缘。



轮廓边的检测根据定义就是，设V为视向量，N1,N2是边的两个相邻的法向量，当(N1·V)*(N2·V) < 0时，可判断此边为轮廓边缘的一部

有人把这2个邻接面当做1个点输入到着色器中，并作为1条边作为输出，并称这2个邻接面为退化四边形。

目前手游中最常见的背面挤出方法只能提供最有限的轮廓边，难以风格化，可以控制颜色和宽度，性能很好。
而主流离线方案质量可以非常高，如Pencil+、Blender FreeStyle等。可以非常自由地进行风格化，最典型的就是线段起始到结束的宽度控制以模拟笔触，但价格和耗时也很高



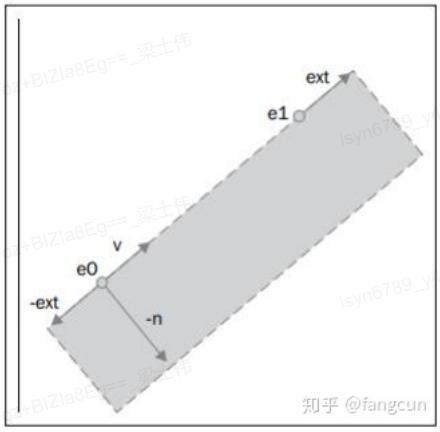
機械的で均一なラインから、オブジェクトからはみ出したラフスケッチのような表現、漫画のような「入り」「抜き」のある抑揚ラインまで、その表現力は多彩です。オブジェクトどうしの交差、マテリアルの境界、オブジェクト背後の陰線など、その詳ン描画箇所の設定はあらゆるシーンに対応します。

根据这种方法的描边在预处理阶段需要将Mesh转化为退化四边形的结构
把复杂对象A当做简单对象B来简化运算，有一个形象的名字叫作“退化对象A”。这里是把四边形当做顶点来简化运算，所以这个（包含点数据）四边形就叫“退化四边形”。因此退化四边形既可叫边，可叫2个三角面，也可叫4个顶点。

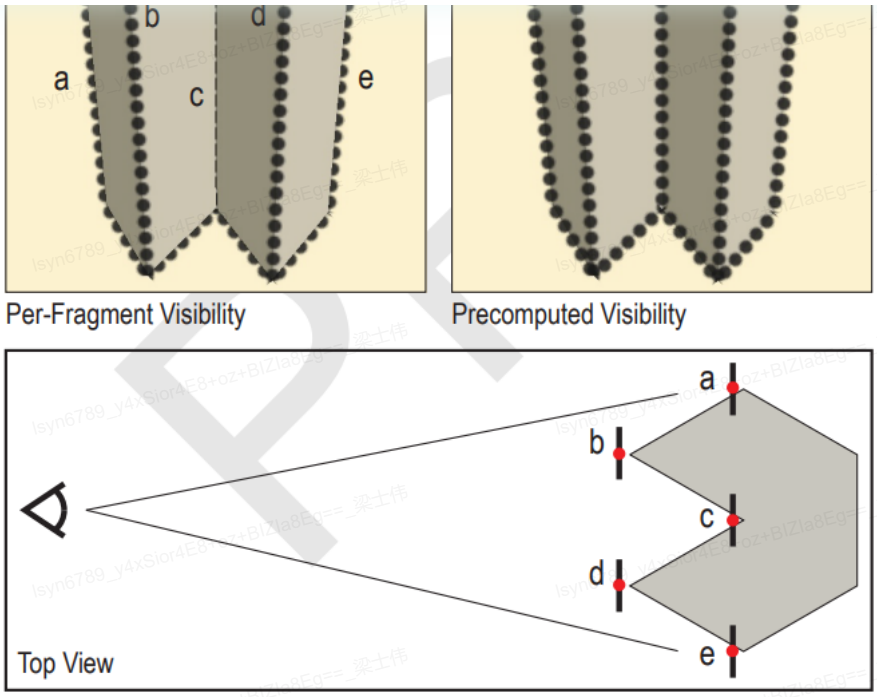
```
//退化四边形
[System.Serializable]
public struct DegradedRectangle {
    public int vertex1;// 构成边的顶点1的索引
    public int vertex2;// 构成边的顶点2的索引
    public int triangle1_vertex3;// 边所在三角面1的顶点3索引
    public int triangle2_vertex3;// 边所在三角面2的顶点3索引
}

//用边的2点来标识退化四边形，可方便去重，注意要重载==和!=
public class CustomLine {
    public Vector3 point1;
    public Vector3 point2;
    public DegradedRectangle degraded_rectangle;
}

// 遍历Mesh.triangles来找到所有退化四边形，要求无重复
var custom_lines = new List<CustomLine>();
int length = triangles.Length / 3;
for (int i = 0; i < length; i++) {
    int vertex1_index = triangles[i * 3];
    int vertex2_index = triangles[i * 3 + 1];
    int vertex3_index = triangles[i * 3 + 2];

    AddCustomLine(vertex1_index, vertex2_index, vertex3_index, vertices, custom_lines);//添加三角图元vertex1和ver
退化四边形（或叫边）
    AddCustomLine(vertex2_index, vertex3_index, vertex1_index, vertices, custom_lines);//添加三角图元v        lver
退化四边形（或叫边）
    AddCustomLine(vertex3_index, vertex1_index, vertex2_index, vertices, custom_lines);//添加三角图元v        lver
退化四边形（或叫边）
}

degraded_rectangles = new List<DegradedRectangle>(custom_lines.Count);
for (int i = 0; i < custom_lines.Count; i++) {
    degraded_rectangles.Add(custom_lines[i].degraded_rectangle);
}
```

```
    int vertex1;
    int vertex2;
    int triangle1_vertex3;
    int triangle2_vertex3;
};

StructuredBuffer<Line> _DegradedRectangles;

Line _line = _DegradedRectangles[id];
float4 vertex1 = float4(_Vertices[_line.vertex1], 1.0f);
float4 vertex2 = float4(_Vertices[_line.vertex2], 1.0f);
float3 vertex1_normal = _Normals[_line.vertex1];
float3 vertex2_normal = _Normals[_line.vertex2];
float4 triangle1_vertex3 = float4(_Vertices[_line.triangle1_vertex3], 1.0f);
float4 center_point = (vertex1 + vertex2 + triangle1_vertex3) / 3.0f;

bool is_edge = 1;
if (_line.triangle2_vertex3 > 0) { // 非边界边
    float4 triangle2_vertex3 = float4(_Vertices[_line.triangle2_vertex3], 1.0f);

    float3 v1 = vertex2 - vertex1;
    float3 v2 = triangle1_vertex3 - vertex1;
    float3 v3 = triangle2_vertex3 - vertex1;

    float3 face1Normal = cross(v1, v2);
    float3 face2Normal = cross(v3, v1);

    bool is_outline = !step(0, dot(face1Normal, view_dir) * dot(face2Normal, view_dir));
    bool is_crease = step(pow(dot(face1Normal, face2Normal) / cos(1.0472f), 2), dot(face1Normal, face1Normal) *
dot(face2Normal, face2Normal));

    is_edge = is_outline | is_crease;
}
```

现阶段问题的难点在于渲染阶段的ZTest，ZTest难在如何稳定地渲染已经提取出来的边而不抖动(深度测试裁剪)。
现有的边渲染方案有
使用几何着色器绘制轮廓线即延法线方向挤出，详情见参考文献[3]



一种是Spine Test方案, 详情见参考文献[4]



采用Ray Tracing方案, 详情见参考文献[5]

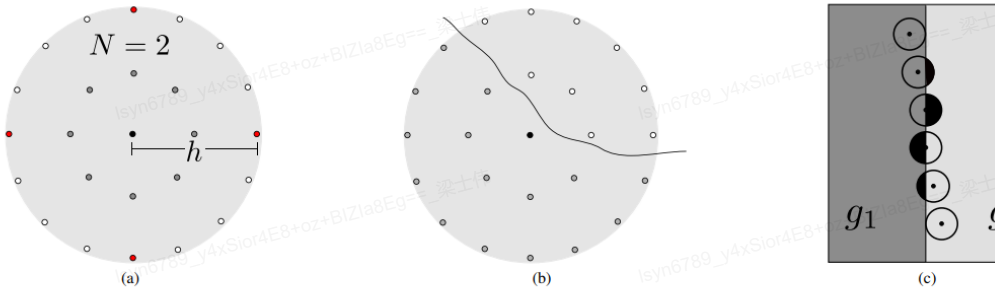


Figure 4: *The details of how ray stencils work. (a) The ray stencil $D_h^N(s)$ samples a disc of radius h around the sample ray posit (black). The samples lie in N concentric circles (dark gray, light gray/red) about s, where N is the* quality parameter *(in this example, N for two rings of samples). The largest circle has radius h. The red samples indicate a finite difference stencil that can be used to me gradients in searching for crease edges. (b) A ray stencil being used to measure foreign primitive area. s strikes some primitive belo black line, and a different primitive lies above the line. The stencil (which excludes s) contains twenty four rays ($M = 24$), and nine of strike a foreign primitive ($m = 9$). Using the linear edge strength metric (Equation 2), which measures how close to* half *of the samples a different primitive, we have $e_s = e_M(9) = 62.5\%$. (c) In the limit as $N \to \infty$, a ray stencil becomes a circular disc, minus its c point. The disc moves across a primitive boundary, from geometry ID g_1 to g_2, acting as an* area indicator *for the portion of the filter tha on a foreign primitive (shaded black, the foreign primitive is g_2 for the top three discs, and g_1 for the bottom three). For the six "snaps shown in this example, the disc moves a distance in screen space of 2h, where h is the disc radius. The foreign area increases from ze one-half, "flips" to the left side as the center of the filter crosses the boundary, and then decreases back to zero. The foreign primitive a used to define an edge strength, which in turn is used to render feature lines.*

4 结语

市面上绝大多数的卡通渲染游戏在描边方面只采用了普通的描边方案，日厂所使用的后处理、顶点色、本村线，可能已经是目前三渲二描边的最佳方案了，但是非常依赖美术，尤其是本村线，国内因其制作成本极高而基本不见使用，而一些日厂却把本村线玩得炉火纯青。虽然更加高级的描边方案在很早就已经被提出，并且这种基于空间的描边极大程度的得到模型丰富的描边细节，可惜在市面[illegible]并不为知。实际中在非真实感上用得很多的“背面法”在很多文献上都有提及，但“背面法”是没办法把模型的所有边缘都获取到[illegible]方法在文献上被记载只能得到模型的轮廓边缘。但是某哈游公司确实在自家的游戏中使用到了多种边缘并结合了其它的卡通渲染方法[illegible]渲染将同行远远甩开

5 参考文献

- [1] Unreal Engine 4 Toon Outlines Tutorial
- [2]基于图像处理的轮廓线渲染
- [3]使用几何着色器绘制轮廓线
- [4]Two Fast Methods for High-Quality Line Visibility
- [5]Ray Tracing NPR-Style Feature Lines

快来成为第一个打赏的人吧~

全部评论 0

请输入评论内容

还可以输入

(可添加1个视频+5张图片) 匿名

最热 最新

暂无评论

加载完毕,没有更多了